JOINT PHYSICS SCALE FACTOR CLASS

Introduction

The Joint Physics scale factor class `JPScaleFactor` is part of the `JointPhysics` package. It can either be built as a standard cvs package or as a standalone library. `JPScaleFactor` replaces the old scale factor classes.

The purpose of this class is to provide a framework of using the blessed Joint Physics scale factors without having to modify the code to update the scale factors. To achieve this, the `JPScaleFactor` class uses a text file to define the scale factors. This text file is usually created from an Excel spreadsheet, `JPScaleFactor.xls`.

JPScaleFactor will calculate luminosity-weighted averages, given luminosity profiles. The text file provided by Joint Physics provides a luminosity-weighted average for all run periods (run `0`). It is also possible, for example, to have the *Period 8* MC use a luminosity-weighted average of the scale factors for *Periods 8 - 12*.

Currently, the `JPScaleFactor` both provides *flat* scale factors (*e.g.,* that do not depend on any variables such as transverse energy) as well as scale factors that depend on upto three variables.

Building the `JointPhysics` Package

If you are using the standard CDF software environment, simply add the package and build it.

```
unix> cd myRelease
unix> addpkg JointPhysics jp080108
unix> gmake JointPhysics.nobin
```

If you want to build it outside the CDF software environment (*e.g.,* top analysis framework), it is also quite simple.

```
unix> cd myAnaDir
unix> ln -s ../JointPhysics/JointPhysics include/JointPhysics
unix> cvs co -r jp080108 JointPhysics
unix> cd JointPhysics
unix> rm -f GNUmakefile; mv Makefile.standalone Makefile
unix> gmake
```

Note: When you compile the package, you will see *"errors"* (they are really warnings) from `rootcint` like:

```
 <**rootcint**> JPScaleFactorFunc_linkdef.h
 Error: *** Datamember JPScaleFactorFunc::m_func1Ptr: no size indication!
 Error: *** Datamember JPScaleFactorFunc::m_func1Ptr: pointer to fundamental type
  (need manual intervention)
```

`rootcint` does not understand function pointers. This does not indicate a problem and can be safely ignored.

Using `JPScaleFactor` Class

The scale factor class is designed to be used both inside of Root as well as in standalone executables. `JPScaleFactor` is a class that reads in a text file of scale factors and efficiencies. Once loaded, you simply set a run number and choose a scale factor.

```
   // create a new object
   JPScaleFactor scale ("JPScaleFactor.txt");
   // or JPScaleFactor scale; scale.loadFile  ("JPScaleFactor.txt");
   // set a run number
   JPScaleFactor::setRunNumber (136802);
   // get 'TCEM_idreco' scale factor, either
   cout << "Method 1: 'TCEM_idreco' scale factor: "
        << scale.value("TCEM_idreco")
        << endl;
   // or
   int TCEM_idreco_index = scale.index("TCEM_idreco");
   cout << "Method 2: 'TCEM_idreco' scale factor: "
        << scale.value (TCEM_idreco_index)
        << endl;
```

To use a functional form

```
    // use trigger which depends on two variables (e.g. METPEM)
    double met = 35;
    double et  = 18;
    cout << "METPEM_trig scale factor " << scale.func2 ("METPEM_trig", met, et) << endl;
```

Note that `func1` is called if it depends on only 1 variable, `func2` if it depends on two, and `func3` if it depends on three variables.

To set a single (flat) scale factor shift up or down

```
// move 'TCEM_idreco' down one sigma
scale.setShift ("TCEM_idreco", -1.);
```

To set a single function scale factor shift up or down

```
// move 'METPEM_trig' down one sigma
scale.setFuncShift ("METPEM_trig", -1.);
```

To move all scale factors (flat and functions) matching a string up 1 sigma

```
// move all '_idreco' down one sigma
scale.setShiftContaining ("_idreco", 1.);
```

To reset all scale factors to their nominal values (*i.e.,* shift of 0),

```
// reset all shifts
scale.resetAllShifts();
```

To print out all scale factors that have been loaded in

```
// Print everything
std::cout << scale << std::endl;
```

That's most of the functionality that you need. You can also look at `jpsf_example.C` that uses more options and at the `JPScaleFactor.hh` for more documentation on the features.

`JPScaleFactor` Text File and Excel Spreadsheet

`JPScaleFactor.txt` is a specially formatted text file that is created from an Excel spreadsheet, `JPScaleFactor.xls`. These files consists of three parts:

1. Scale factors
2. Luminosity profiles
3. Applying luminosity weights

To convert from the Excel spreadsheet to the text file, you use `makeJPScaleFactorTxt.pl`.

1. Open `JPScaleFactor.xls` in Excel. Since the spreadsheet now has multiple sheets, you'll want to save each one (that has numbers) as a text file
2. File -> Save As - Save as type: Text (Tab delimited) - File name: *e.g.,* `scalefactor.txt` or `turnon.txt`
3. Copy these text files to fcdflnx4
4. `unix> makeJPScaleFactorTxt.pl scalefactor.txt turnon.txt`
5. Use `JPScaleFactor.txt`

Lines in `JPScaleFactor.txt` that begin with a hash (`#`) character are treated as comments and ignored.

Verify Format of `JPScaleFactor.txt`

After creating `JPScaleFactor.txt`, you can use `JointPhysics/examples/verifyJPSF.C` to make sure that the text file is properly formatted. Note that if you used the standard CDF Makefile (as opposed to `Makefile.standalone`), then you will need to edit this file changing the line (around line 123):

```
gSystem->Load("../../shlib/libJointPhysics");
```

to

```
gSystem->Load("../../shlib/Linux2_SL-GCC_3_4/libJointPhysics");
```

To run this test,

```
root -l -q 'verifyJPSF.C("directory/WhereIhave/JPScaleFactor.txt")' >& out
```

If your copy of `out` looks like the example provided, there are not formatting errors.

Scale Factors

Scale factors are defined as a name and then groups of four numbers, specifically *starting run number*, *ending run number*, *scale factor value*, and *scale factor error*.

For example, we can define the `TCEM_trig` scale factor:

`TCEM_trig` with four run periods defined on one line

```
TCEM_trig         138425 186598 0.962  0.007   190697 203799 0.976  0.006   203819 212133 0.979  0.004   217990 222426 0.959  0.007
```

`TCEM_trig` with four run periods, each on its own line

```
TCEM_trig         138425 186598 0.962  0.007
TCEM_trig         190697 203799 0.976  0.006
TCEM_trig         203819 212133 0.979  0.004
TCEM_trig         217990 222426 0.959  0.007
```

Luminosity Profiles

A luminosity profile is a named object that defines different integrated luminosities for different run periods. First, to describe a run period, all you need to do is provide any run in the run range (I recommend using the first run number). A luminosity profile is defined by `- lum` followed by a a name and then pairs of numbers, *run number* and *integrated luminosity*.

```
- lum standard    138425 200 190697 400 203819 600  217990 800
- lum higher_runs 203819 600 217990 800
```

defines a luminosity profile called `standard` with four run ranges. The first, starting at run `138425` has `200` pb^{-1} of integrated luminosity. The second, starting at run `190697` has `400` pb^{-1} of integrated luminosity, etc. A luminosity profile **must** be defined on a single line.

Applying Luminosity Weights

Here, we simply tell a scale factor that a certain run range (again, described by a single run contained inside the run range) should actually use a luminosity-weighted average. This is done with a line that starts `- weight lumiName sfname1 sfname2`. This can be split over several lines. If we want to set the luminosity average for the whole run period, we always use **run number** `0`.

If I want to set the average for the `TCEM` trigger efficiency and ID scale factor, I would

```
- weight standard      0 TCEM_trig TCEM_id
```

If I was reweighting MC using runs 203819 to 212133 to represent runs 203819 to 222426, I could do this by:

```
- weight higher_runs  203819 TCEM_trig TCEM_id
```

If I want to hook up a functional scale factor, `MET_trig` that I want using the step function `step_1`. In this example, I want `step_1` to take `5` parameters, the starting and ending run numbers `100000` and `199999`, and the list the five parameters:`0.2`, `0.02`, `20`, `0.9`, and `0.1`. See below for more information on the `step_1` function.

```
- func MET_trig step_1 5 100000 199999 0.2 0.02 20 0.9 0.1
```

Rules for `JPScaleFactor.xls`

Here are the rules for making `JPScaleFactor.xls`

- Blank lines are o.k.
- All "names" can only contain letters, numbers, underscore '_', and dash '-'. **No Spaces**.
- Non-empty lines that have a blank first column "A" are scale factor lines.
  - The first (non-empty) column is the name.
  - The next four columns correspond to the information for a run period: *starting run number*, *ending run number*, *scale factor value*, and *scale factor error*.
  - You can place as many run periods on a single line as you wish.
  - You can break up a single scale factor (*e.g.,* `TCEM_id` into as many lines as you wish).
  - Run periods **must** be disjoint.
- Command lines (*e.g.,* `- lum` and `- weight`)
  - Must start with the command in the first column.
  - Everything after the command can appear either in a single column separated by spaces or in multiple columns
- All other lines are treated as comment lines.

Scale Factors with Functional Forms

The idea behind the functional forms is quite simple. In `JPScaleFactor.xls`, you start with `- func`, specify the name of the scale factor (*e.g.,* `MET_trig`), and then you tell it which function you want that describes the functional form (*e.g.,* `step_1`). You then need to tell it how many parameters you need to pass to this function and then list the starting run number, the ending run number, and the number of parameters you specified.

All functional forms that are used by the default `JPScaleFactor.xls` are already hooked up. If you want to use another functional form, all you need to do is tell the code. Let's use the example of the two functions below. All functions must return both a *value* and an *error*.

```
// two different functions: the first for two parameter functional
forms.  The second for three parameter functional forms.
void twoParam (double &value, double &error,
               const JPScaleFactorFunc::DVec &paramVec,
               double alpha, double beta);
void threeParam (double &value, double &error,
                 const JPScaleFactorFunc::DVec &paramVec,
                 double alpha, double beta, double gamma);
```

In this case, I would register these functions by calling `JPScaleFactorFunc::add1ParamFunction`, `JPScaleFactorFunc::add2ParamFunction`, or `JPScaleFactorFunc::add3ParamFunction` for 1, 2, and 3 parameter functional forms respectively. This function takes:

1. The name which the will be used in `JPScaleFactor.xls`. It must end in `_1`, `_2`, or `_3` which tells it how many parameters it uses.
2. The pointer to the function in question.
3. The minimum number of parameters that this function requires.
4. (Optionally) A text description of this function that is printed out if the scale factor is printed.

In the case of the two functions above, you would hook them up by

```
// register the two and three parameter functions
JPScaleFactorFunc::add2ParamFunction ("twoParam_2",  // name of function
                                      &twoParam, // pointer to function
                                      3,             // minimum number of parameters
```

```
                                        "Silly 2 parameter function");
   JPScaleFactorFunc::add3ParamFunction ("threeParam_3",
                                         &threeParam,
                                         4,
                                         "Silly 3 parameter function");
```

The functions in question must modify both the `value` and `error`. Here are working (but silly) examples

```
void twoParam (double &value, double &error,
               const JPScaleFactorFunc::DVec &paramVec,
               double alpha, double beta)
{
   // When this function was registered, I put in '3' as the minimum
   // number of parameters.  This means that in order to get to this
   // point, paramVec must have at least 3 entries.  These are referenced
   // as paramVec.at(0) up to paramVec.at(2).  If you try to use an
   // index that is not valid, your code will segfault.
   value =
      1.  * paramVec.at(0) * alpha +
      10. * paramVec.at(1) * beta;
   error = value * paramVec.at(2);
}

void threeParam (double &value, double &error,
                 const JPScaleFactorFunc::DVec &paramVec,
                 double alpha, double beta, double gamma)
{
   value =
      1.   * paramVec.at(0) * alpha +
      10.  * paramVec.at(1) * beta  +
      100. * paramVec.at(2) * gamma;
   error = value * paramVec.at(3);
}
```

Two last points: First, remember that you can load multiple files into a single `JPScaleFactor` object. This means that you can load in the standard `JPScaleFactor.txt` and then load in your own file. Second, see `jpsf example.C` for a complete example.

Functions Already Hooked up

At this time, only the 1-D step function `step_1` is hooked up by default. This function takes 2, 5, 8, ... 3 * N - 1 parameters.

- `0.9 0.1` - will always have a value of `0.9` and an error of `0.1`
- `0. 0. 15 0.9 0.1 20 0.99 0.15` - will return `0` for both the value and the error if the paramter is less than `15`, `0.9` for the value and `0.1` for the error if the value is less than `20`, otherwise `0.99` and `0.15` for the value and error.

Last modified: Thu Jan 15 07:35:19 CST 2009